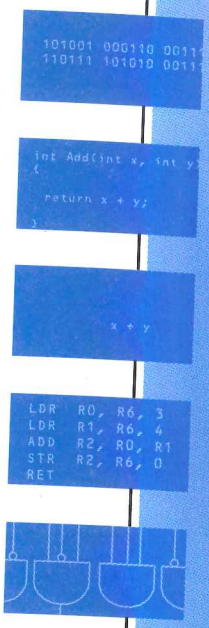
chapter

13



Control Structures

13.1 Introduction

In Chapter 6, we introduced our top-down problem-solving methodology where a problem is systematically refined into smaller, more detailed subtasks using three programming constructs: the sequential construct, the conditional construct, and the iteration construct.

We applied this methodology in the previous chapter to derive a simple C program that calculates network transfer time. The problem's refinement into a program only required the use of the sequential construct. For transforming more complex problems into C programs, we will need a way to invoke the conditional and iteration constructs in our programs. In this chapter, we cover C's version of these two constructs.

We begin this chapter by describing C's conditional constructs. The `if` and `if-else` statements allow us to conditionally execute a statement. After conditional constructs, we move on to C's iteration constructs: the `for`, the `while`, and the `do-while` statements, all of which allow us to express loops. With many of these constructs, we will present the corresponding LC-3 code generated by our hypothetical LC-3 C compiler to better illustrate how these constructs behave at the lower levels. C also provides additional control constructs, such as the `switch`, `break`, and `continue` statements, all of which provide a convenient way to represent some particular control tasks. We discuss these in Section 13.5. In the final part of the chapter, we'll use the top-down problem-solving methodology to solve some complex problems involving control structures.

13.2 Conditional Constructs

Conditional constructs allow a programmer to select an action based on some condition. This is a very common programming construct and is supported by every useful programming language. C provides two types of basic conditional constructs: `if` and `if-else`.

13.2.1 The `if` Statement

The `if` statement is quite simple. It performs an action if a condition is true. The action is a C statement, and it is executed only if the condition, which is a C expression, evaluates to a nonzero (logically true) value. Let's take a look at an example.

```
if (x <= 10)
   y = x * x + 5;
```

The statement `y = x * x + 5;` is only executed if the expression `x <= 10` is nonzero. Recall from our discussion of the `<=` operator (the less than or equal to operator) that it evaluates to 1 if the relationship is true, 0 otherwise.

The statement following the condition can also be a *compound statement*, or *block*, which is a sequence of statements beginning with an open brace and ending with a closing brace. Compound statements are used to group one or more simple statements into a single entity. This entity is itself equivalent to a simple statement. Using compound statements with an `if` statement, we can conditionally execute several statements on a single condition. For example, in the following code, both `y` and `z` will be modified if `x` is less than or equal to 10.

```
if (x <= 10) {
   y = x * x + 5;
   z = (2 * y) / 3;
}
```

As with all statements in C, the format of the `if` statement is flexible. The line breaks and indentation used in the preceding example are features of a popular style for formatting an `if` statement. It allows someone reading the code to quickly identify the portion that executes if the condition is true. Keep in mind that the format does not affect the behavior of the program. Even though the following code is indented like the previous code, it behaves differently. The second statement `z = (2 * y) / 3;` is not associated with the `if` and will execute regardless of the condition.

```
if (x <= 10)
   y = x * x + 5;
   z = (2 * y) / 3;
```

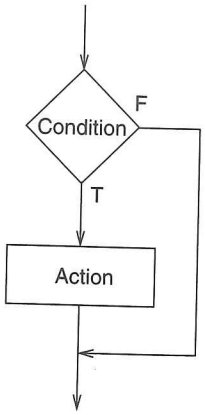


Figure 13.1 The C `if` statement, pictorially represented

Figure 13.1 shows the control flow of an `if` statement. The diagram corresponds to the following code:

```
if (condition)
   action;
```

Syntactically, the condition must be surrounded by parentheses in order to enable the compiler to unambiguously separate the condition from the rest of the `if` statement. The action must be a simple or compound statement.

Here are more examples of `if` statements demonstrating programming situations where this decision construct might be useful.

```
if (temperature <= 0)
   printf("At or below freezing point.\n");

if ('a' <= key && key <= 'z')
   numLowerCase++;

if (current > currentLimit)
   blownFuse = 1;

if (loadMAR & clock)
   registerMAR = bus;

if (month == 4 || month == 6 || month == 9 || month == 11)
   printf("The month has 30 days\n");

if (x = 2)    /* This condition is always true.     */
   y = 5;     /* The variable y will always be 5    */
```

The last example in the preceding code illustrates a very common mistake made when programming in C. (Sometimes even expert C programmers make this mistake. Good C compilers will warn you if they detect such code.) The condition uses the assignment operator = rather than the equality operator, which causes the value of x to change to 2. This condition is always true: expressions containing the assignment operator evaluate to the value being assigned (in this case, 2). Since the condition is always nonzero, y will always get assigned the value 5 and x will always be assigned 2.

Even though they look similar at first glance, the following code is a "repaired" version of the previous code.

```
if (x == 2)
   y = 5;
```

Let's look at the LC-3 code that is generated for this code, assuming that x and y are integers that are locally declared. This means that R5 will point to the variable x and R5 - 1 will point to y.

```
          LDR  R0, R5, #0   ; load x into R0
          ADD  R0, R0, #-2  ; subtract 2 from x
          BRnp NOT_TRUE     ; If condition is not true,
                            ; then skip the assignment

          AND  R0, R0, #0   ; R0 <- 0
          ADD  R0, R0, #5   ; R0 <- 5
          STR  R0, R5, #-1  ; y = 5;

NOT_TRUE  :                 ; the rest of the program
          :
```

Notice that it is most straightforward for the LC-3 C compiler to generate code that tests for the opposite of the original condition (x not equal to 2) and to branch based on its outcome.

The if statement is itself a statement. Therefore, it is legal to *nest* an if statement as demonstrated in the following C code. Since the statement following the first if is a simple statement (i.e., composed of only one statement), no braces are required.

```
if (x == 3)
   if (y != 6) {
      z = z + 1;
      w = w + 2;
   }
```

The inner `if` statement only executes if `x` is equal to 3. There is an easier way to express this code. Can you do it with only one `if` statement? The following code demonstrates how.

```
if ((x == 3) && (y != 6)) {
      z = z + 1;
      w = w + 2;
   }
```

13.2.2 The `if-else` Statement

If we wanted to perform one set of actions if a condition were true and another set if the same condition were false, we could use the following sequence of `if` statements:

```
if (temperature <= 0)
   printf("At or below freezing point.\n");

if (temperature > 0)
   printf("Above freezing.\n");
```

Here, a single message is printed depending on whether the variable `temperature` is below or equal to zero or if it is above zero. It turns out that this type of conditional execution is a very useful construct in programming. Since expressing code in the preceding way can be a bit cumbersome, C provides a more convenient construct: the `if-else` statement.

The following code is equivalent to the previous code segment.

```
if (temperature <= 0)
   printf("At or below freezing point.\n");
else
   printf("Above freezing.\n");
```

Here, the statement appearing immediately after the `else` keyword executes only if the condition is false.

The flow diagram for the `if-else` is shown in Figure 13.2. The figure corresponds to the following code:

```
if (condition)
   action_if;
else
   action_else;
```

The lines `action_if` and `action_else` can correspond to compound statements and thus consist of multiple statements, as in the following example.

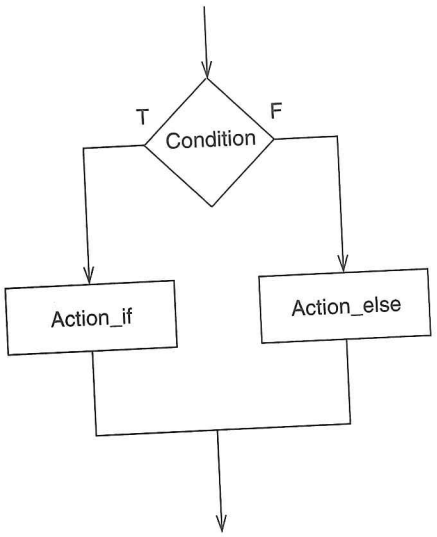


Figure 13.2 The C `if-else` statement, pictorially represented

```
if (x) {
   y++;
   z--;
}
else {
   y--;
   z++;
}
```

If the variable `x` is nonzero, the `if`'s condition is true, `y` is incremented, and `z` decremented. Otherwise, `y` is decremented and `z` incremented. The LC-3 code generated by the LC-3 C compiler is listed in Figure 13.3. The three variables `x`, `y`, and `z` are locally declared integers.

We can connect conditional constructs together to form a longer sequence of conditional tests. The example in Figure 13.4 shows a complex decision structure created using the `if` and `if-else` statements. No other control structures are used. This program gets a number of a month from the user and displays the number of days in that month.

At this point, we need to mention a C syntax rule for associating `if`s with `else`s: An `else` is associated with the closest unassociated `if`. The following example points out why this is important.

```
if (x != 10)
   if (y > 3)
      z = z / 2;
   else
      z = z * 2;
```

```
1          LDR  R0, R5, #0    ;   load the value of x
2          BRz  ELSE          ;   if x equals 0, perform else part
3
4          LDR  R0, R5, #-1   ;   load y into R0
5          ADD  R0, R0, #1
6          STR  R0, R5, #-1   ;   y++;
7
8          LDR  R0, R5, #-2   ;   load z into R0
9          ADD  R0, R0, #-1
10         STR  R0, R5, #-2   ;   z--;
11         BR DONE
12
13   ELSE: LDR  R0, R5, #-1   ;   load y into R0
14         ADD  R0, R0, #-1
15         STR  R0, R5, #-1   ;   y--;
16
17         LDR  R0, R5, #-2   ;   load z into R0
18         ADD  R0, R0, #1
19         STR  R0, R5, #-2   ;   z++;
20   DONE: :
21         :
```

Figure 13.3 The LC-3 code generated for an `if-else` statement

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int month;
6
7    printf("Enter the number of the month: ");
8    scanf("%d", &month);
9
10   if (month == 4 || month == 6 || month == 9 || month == 11)
11     printf("The month has 30 days\n");
12   else if (month == 1 || month == 3 || month == 5  ||
13            month == 7 || month == 8 || month == 10 || month == 12)
14     printf("The month has 31 days\n");
15   else if (month == 2)
16     printf("The month has either 28 days or 29 days\n");
17   else
18     printf("Don't know that month\n");
19 }
```

Figure 13.4 A program that determines the number of days in a month

Without this rule, it would not be clear whether the `else` should be paired with the *outer* `if` or the *inner* `if`. For this situation, the rule states that the `else` is coupled with the inner `if` because it is closer than the outer `if` and the inner `if` statement has not already been coupled to another `else` (i.e., it is unassociated). The code is equivalent to the following:

```
if (x != 10) {
   if (y > 3)
      z = z / 2;
   else
      z = z * 2;
}
```

Just as parentheses can be used to modify the order of evaluation of expressions, braces can be used to associate statements. If we wanted to associate the `else` with the outer `if`, we could write the code as

```
if (x != 10) {
   if (y > 3)
      z = z / 2;
}
else
   z = z * 2;
```

Before we leave the `if-else` statement for bigger things, we present a very common use for the `if-else` construct. The `if-else` statement is very handy for checking for bad situations during program execution. We can use it for error checking, as shown in Figure 13.5. This example performs a simple division based on two numbers scanned from the keyboard. Because division by 0 is undefined, if the user enters a 0 divisor, a message is displayed indicating the result cannot be generated. The `if-else` statement serves nicely for this purpose.

Notice that the nonerror case appears in the `if-else` statement first and the error case second. Although we could have coded this either way, having the common, nonerror case first provides a visual cue to someone reading the code that the error case is the uncommon one.

13.3 Iteration Constructs

Being able to iterate, or repeat, a computation is part of the power of computing. Almost all useful programs perform some form of iteration. In C, there are three iteration constructs, each a slight variant of the others: the `while` statement, the `for` statement, and the `do-while` statement.

13.3.1 The `while` Statement

We begin by describing C's simplest iteration statement: the `while`. A `while` loop executes a statement repeatedly *while* a condition is true. Before each iteration

```
#include <stdio.h>

int main()
{
  int dividend;
  int divisor;
  int result;

  printf("Enter the dividend: ");
  scanf("%d", &dividend);

  printf("Enter the divisor: ");
  scanf("%d", &divisor);

  if (divisor != 0) {
    result = dividend / divisor;
    printf("The result of the division is %d\n", result);
  }
  else
    printf("A divisor of zero is not allowed\n");
}
```

Figure 13.5 A program that has error-checking code

of the statement, the condition is checked. If the condition evaluates to a logical true (nonzero) value, the statement is executed again.

In the following example program, the loop keeps iterating while the value of variable `x` is less than 10. It produces the following output:

```
0 1 2 3 4 5 6 7 8 9
```

```
#include <stdio.h>

int main()
{
   int x = 0;

   while (x < 10) {
      printf("%d ", x);
      x = x + 1;
   }
}
```

The `while` statement can be broken down into two components. The `test` condition is an expression used to determine whether or not to continue executing the loop.

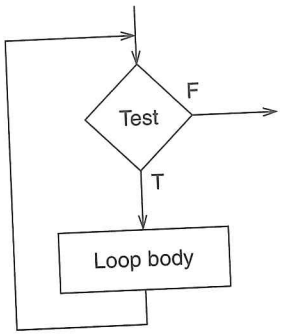


Figure 13.6 The C `while` statement, pictorially represented

```
while (test)
   loop_body;
```

It is tested before each execution of the `loop_body`. The `loop_body` is a statement that expresses the work to be done within the loop. Like all statements, it can be a compound statement.

Figure 13.6 shows the control flow using the notation of systematic decomposition. Two branches are required: one conditional branch to exit the loop and one unconditional branch to loop back to the test to determine whether or not to execute another iteration.

The LC-3 code generated by the compiler for the `while` example that counts from 0 to 9 is listed in Figure 13.7.

The `while` statement is useful for coding loops where the iteration process involves testing for a *sentinel* condition. That is, we don't know the number of iterations beforehand but we wish to keep looping until some event (i.e., the

```
1                AND  R0, R0, #0    ;   clear out R0
2                STR  R0, R5, #0    ;   x = 0;
3
4                ; while (x < 10)
5     LOOP:      LDR  R0, R5, #0    ;   perform the test
6                ADD  R0, R0, #-10
7                BRpz DONE          ;   x is not less than 10
8
9                ; loop body
10               :
11               <code for calling the function printf>
12               :
13               LDR  R0, R5, #0    ;   R0 <- x
14               ADD  R0, R0, #1    ;   x + 1
15               STR  R0, R5, #0    ;   x = x + 1;
16               BR   LOOP          ;   another iteration
17    DONE:      :
18               :
```

Figure 13.7 The LC-3 code generated for a `while` loop that counts to 9

```
1  #include <stdio.h>
2
3  int main()
4  {
5    char echo = 'A';    /* Initialize char variable echo */
6
7    while (echo != '\n') {
8      scanf("%c", &echo);
9      printf("%c", echo);
10   }
11 }
```

Figure 13.8 Another program with a simple `while` loop

sentinel) occurs. For example, when we wrote the character counting program in Chapters 5 and 7, we created a loop that terminated when the sentinel EOT character (a character with ASCII code 4) was detected. If we were coding that program in C rather than LC-3 assembly language, we would use a `while` loop. The program in Figure 13.8 uses the `while` statement to test for a sentinel condition. Can you determine what this program does without executing it?[1]

We end our discussion of the `while` statement by pointing out a common mistake when using `while` loops. The following program will never terminate because the loop body does not change the looping condition. In this case, the condition always remains true and the loop never terminates. Such loops are called *infinite loops*, and most of the time they occur because of programming errors.

```
#include <stdio.h>

int main()
{
   int x = 0;

   while (x < 10)
      printf("%d ", x);
}
```

13.3.2 The `for` Statement

Just as the `while` loop is a perfect match for a sentinel-controlled loop, the C `for` loop is a perfect match for a counter-controlled loop. In fact, the `for` loop is a special case of the `while` loop that happens to work well when the number of iterations is known ahead of time.

[1] This program behaves a bit differently than you might expect. You might expect it to print out each input character as the user types it in. Because of the way C deals with keyboard I/O, the program does not get any input until the user hits the Enter key. We explain why this is so when dealing with the low-level issues surrounding I/O in Chapter 18.

In its most straightforward form, the `for` statement allows us to repeat a statement a specified number of times. For example,

```
#include <stdio.h>

int main()
{
   int x;

   for (x = 0; x < 10; x++)
      printf("%d ", x);
}
```

will produce the following output. It loops exactly 10 times.

```
0 1 2 3 4 5 6 7 8 9
```

The syntax for the C `for` statement may look a little perplexing at first. The `for` statement is composed of four components, broken down as follows:

```
for (init; test; reinit)
   loop_body;
```

The three components within the parentheses, `init`, `test`, and `reinit`, control the behavior of the loop and must be separated by semicolons. The final component, `loop_body`, specifies the actual computation to be executed in each iteration.

Let's take a look at each component of the `for` loop in detail. The `init` component is an expression that is evaluated before the **first** iteration. It is typically used to initialize variables in preparation for executing the loop.

The `test` is an expression that gets evaluated before *every* iteration to determine if another iteration should be executed. If the `test` expression evaluates to zero, the `for` terminates and the control flow passes to the statement immediately following the `for`. If the expression is nonzero, another iteration of the `loop_body` is performed. Therefore, in the previous code example, the test expression `x < 10` causes the loop to keep repeating as long as `x` is less than 10.

The `reinit` component is an expression that is evaluated at the end of *every* iteration. It is used to prepare (or reinitialize) for the next iteration. In the previous code example, the variable `x` is incremented before each repetition of the loop body.

The `loop_body` is a statement that defines the work to be performed in each iteration. It can be a compound statement.

Figure 13.9 shows the flow diagram of the `for` statement. There are four blocks, one for each of the four components of the `for` statement. There is a conditional branch that determines whether to exit the loop based on the outcome of the `test` expression or to proceed with another iteration. An unconditional

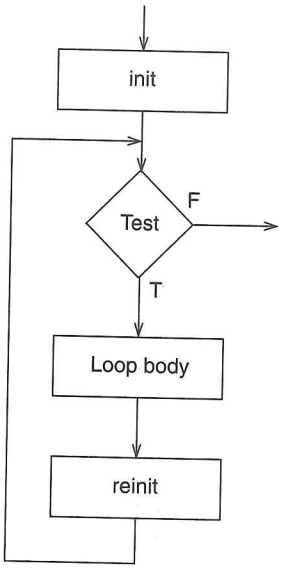


Figure 13.9 The C `for` statement

branch loops back to the `test` at the end of each iteration, after the `reinit` expression is evaluated.

Even though the syntax of a `for` statement allows it to be very flexible, most of the `for` loops you will encounter (or will write) will be of the counter-controlled variety, that is, loops that iterate for a certain number of iterations. Following are some examples of code that demonstrate the counter-controlled nature of `for` loops.

```
/* --- What does the loop output?   --- */
for (x = 0; x <= 10; x++)
   printf("%d ", x);

/* --- What does this one output?   --- */
letter = 'a';

for (c = 0; c < 26; c++)
   printf("%c ", letter + c);

/* --- What does this loop do?      --- */
numberOfOnes = 0;

for (bitNum = 0; bitNum < 16; bitNum++) {
   if (inputValue & (1 << bitNum))
      numberOfOnes++;
}
```

```
1              AND  R0, R0, #0    ;   clear out R0
2              STR  R0, R5, #-1   ;   sum = 0;
3
4              ; init
5              AND  R0, R0, #0    ;   clear out R0
6              STR  R0, R5, #0    ;   init (x = 0)
7
8              ; test
9   LOOP:      LDR  R0, R5, #0    ;   perform the test
10             ADD  R0, R0, #-10
11             BRpz DONE          ;   x is not less than 10
12
13             ; loop body
14             LDR  R0, R5, #0    ;   get x
15             LDR  R1, R5, #-1   ;   get sum
16             ADD  R1, R1, R0    ;   sum + x
17             STR  R0, R5, #-1   ;   sum = sum + x;
18
19             ; reinit
20             LDR  R0, R5, #0    ;   get x
21             ADD  R0, R0, #1
22             STR  R0, R5, #0    ;   x++
23             BR   LOOP
24
25  DONE:      :
26             :
```

Figure 13.10 The LC-3 code generated for a `for` statement

Let's take a look at the LC-3 translation of a simple `for` loop. The program is a simple one: it calculates the sum of all integers between 0 and 9.

```
#include <stdio.h>

int main()
{
   int x;
   int sum = 0;

   for (x = 0; x < 10; x++)
      sum = sum + x;
}
```

The LC-3 code generated by the compiler is shown in Figure 13.10.

The following code contains a mistake commonly made when using `for` loops.

```
sum = 0;
for (x = 0; x < 10; x++);
   sum = sum + x;

printf("sum = %d\n", sum);
printf("x = %d\n", x);
```

What is output by the first `printf`? The answer is `sum = 10`. Why? The second `printf` outputs `x = 10`. Why? If you look carefully, you might be able to notice a misplaced semicolon.

A `for` loop can be constructed using a `while` loop (actually, vice versa as well). In programming, they can be used interchangeably, to a degree. Which construct to use in which situation may seem puzzling at first, but keep in mind the general rule that `while` is best suited for loops that involve sentinel conditions, whereas `for` fits situations where the number of iterations is known beforehand.

Nested Loops

Figure 13.11 contains an example of a `for` where the loop body is composed of another `for` loop. This construct is referred to as a *nested loop* because the inner loop is nested within the outer. In this example, the program prints out a multiplication table for the numbers 0 through 9. Each iteration of the inner loop prints out a single product in the table. That is, the inner loop iterates 10 times for each iteration of the outer loop. An entire row is printed for each iteration of the outer loop. Notice that the `printf` function call contains a special character sequence in its format string. The `\t` sequence causes a tab character to be printed out. The tab helps align the columns of the multiplication table so the output looks neater.

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int multiplicand;    /* First operand of each multiply  */
6    int multiplier;      /* Second operand of each multiply */
7
8    /* Outer Loop */
9    for (multiplicand = 0; multiplicand < 10; multiplicand++) {
10     /* Inner Loop */
11     for (multiplier = 0; multiplier < 10; multiplier++) {
12       printf("%d\t", multiplier * multiplicand);
13     }
14     printf("\n");
15   }
16 }
```

Figure 13.11 A program that prints out a multiplication table

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int sum = 0;                  /* Initial the result variable */
6    int input;                    /* Holds user input            */
7    int inner;                    /* Iteration variables         */
8    int outer;
9
10   /* Get input */
11   printf("Input an integer: ");
12   scanf("%d", &input);
13
14   /* Perform calculation */
15   for (outer = 1; outer <= input; outer++)
16     for (inner = 0; inner < outer; inner++) {
17       sum += inner;
18     }
19
20   /* Output result */
21   printf("The result is %d\n", sum);
22 }
```

Figure 13.12 A program with a nested `for` loop

Figure 13.12 contains a slightly more complex example. The number of iterations of the inner loop depends on the value of `outer` as determined by the outer loop. The inner loop will first execute 0 time, then 1 time, then 2 times, etc. For a challenging exercise based on this example, see Exercise 13.6 at the end of this chapter.

13.3.3 The `do-while` Statement

With a `while` loop, the condition is always evaluated *before* an iteration is performed. Therefore, it is possible for the `while` loop to execute zero iterations (i.e., when the condition is false from the start). There is a slight variant of the `while` statement in C called `do-while`, which always performs at least one iteration. In a `do-while` loop, the condition is evaluated *after* the first iteration is performed. The operation of the `do-while` is demonstrated in the following example:

```
x = 0;
do {
   printf("%d \n", x);
   x = x + 1;
} while (x < 10);
```

Here, the conditional test, `x < 10`, is evaluated at the end of each iteration. Thus, the loop body will execute at least once. The next iteration is performed

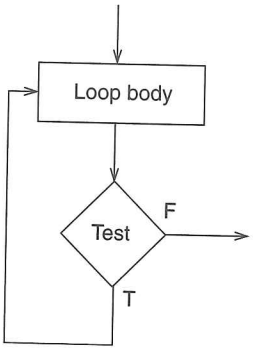


Figure 13.13 The C `do-while` statement

only if the test evaluates to a nonzero value. This code produces the following output:

```
0 1 2 3 4 5 6 7 8 9
```

Syntactically, a `do-while` is composed of two components, exactly like the `while`.

```
do
  loop_body;
while (test);
```

The `loop_body` component is a statement (simple or compound) that describes the computation to be performed by the loop. The `test` is an expression that determines whether another iteration is to be performed.

Figure 13.13 shows the control flow of the `do-while` loop. Notice the slight change from the flow of a `while` loop. The loop body and the test are interchanged. A conditional branch loops back to the top of the loop body, initiating another iteration.

At this point, the differences between the three types of C iteration constructs may seem very subtle, but once you become comfortable with them and build up experience using these constructs, you will more easily be able to pick the right construct to fit the situation. To a large degree, these constructs can be used interchangeably. Stylistically, there are times when one construct makes more sense to use than another—often the type of loop you choose will convey information about the intent of the loop to someone reading your code.

13.4 Problem Solving Using Control Structures

Armed with a new arsenal of control structures, we can attempt to solve more complex programming problems. In this section, we will apply our top-down problem-solving methodology to four problems requiring the use of C control structures.

Being effective at solving programming problems requires that you understand the basic primitives of the system on which you are programming. You will need to invoke them at the appropriate times to solve various programming puzzles. At this point, our list of C primitives includes variables of the three basic types, operators, two decision structures, and three control structures.

13.4.1 Problem 1: Approximating the Value of π

For the first programming problem, we will calculate the value of π using the following series expansion:

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \cdots + (-1)^{n-1}\frac{4}{2n+1} + \cdots$$

The problem is to evaluate this series for the number of terms indicated by the user. If the user enters 3, the program will evaluate $4 - \frac{4}{3} + \frac{4}{5}$. The series is an infinite series, and the more terms we evaluate, the more accurate our approximation of π.

As we did for the problem-solving example in Chapter 12, we first invoke step 0: we select a representation for the data involved in the computation. Since the series deals with fractional numbers, we use the `double` floating point type for any variables directly involved in the series calculation. Given the nature of the computation, this seems clearly to be the best choice.

Now we invoke stepwise refinement to decompose a roughly stated algorithm into a C program. Roughly, we want the program to initialize all data that requires initialization. Then ask the user to input the number of terms of the series to evaluate. Then evaluate the series for the given number of terms. Finally, print out the result. We have defined the problem as a set of sequential constructs. Figure 13.14 shows the decomposition thus far.

Most of the sequential constructs in Figure 13.14 are very straightforward. Converting them into C code should be quite simple. One of the constructs in the figure, however, requires some additional refinement. We need to put a little thought into the subtask labeled *Evaluate series*. For this subtask, we essentially want to *iterate* through the series, term by term, until we evaluate exactly the number of terms indicated by the user. We want to use a counter-controlled iteration construct. Figure 13.15 shows the decomposition. We maintain a counter for the current loop iteration. If the counter is less than the limit indicated by the user, then we evaluate another term. Notice that the refined version of the subtask looks like the flow diagram for a `for` loop.

We are almost done. The only nontrivial subtask remaining is *Evaluate another term*. Notice that all even terms in the series are subtracted, and all odd terms are added. Within this subtask, we need to determine if the particular term we are evaluating is an odd or an even term, and then accordingly factor it into the current value of the approximation. This involves using a decision construct as shown in Figure 13.16. The complete code resulting from this stepwise refinement is shown in Figure 13.17.

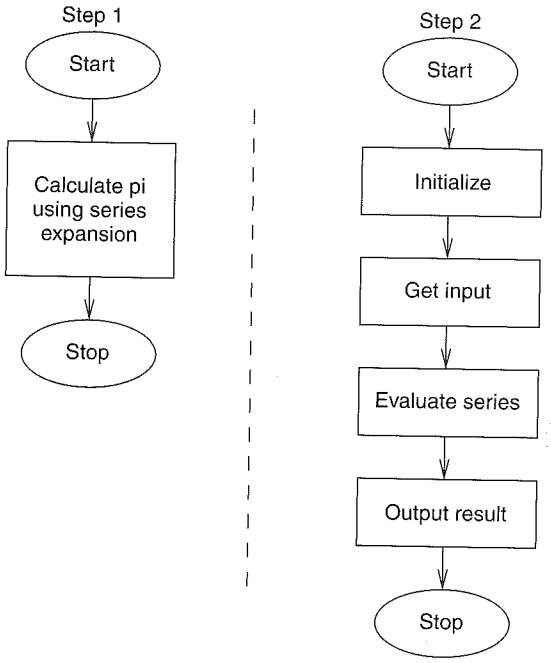


Figure 13.14 The initial decomposition of a program that evaluates the series expansion for π for a given number of terms

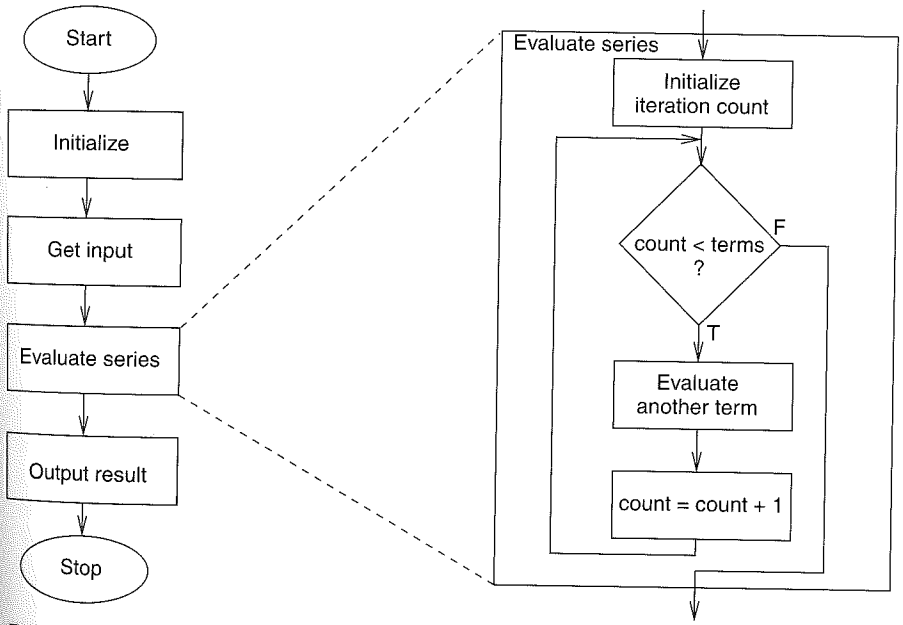


Figure 13.15 The refinement of the subtask Evaluate series into an iteration construct that iterates a given number of times. Within this loop, we evaluate terms for a series expansion for π

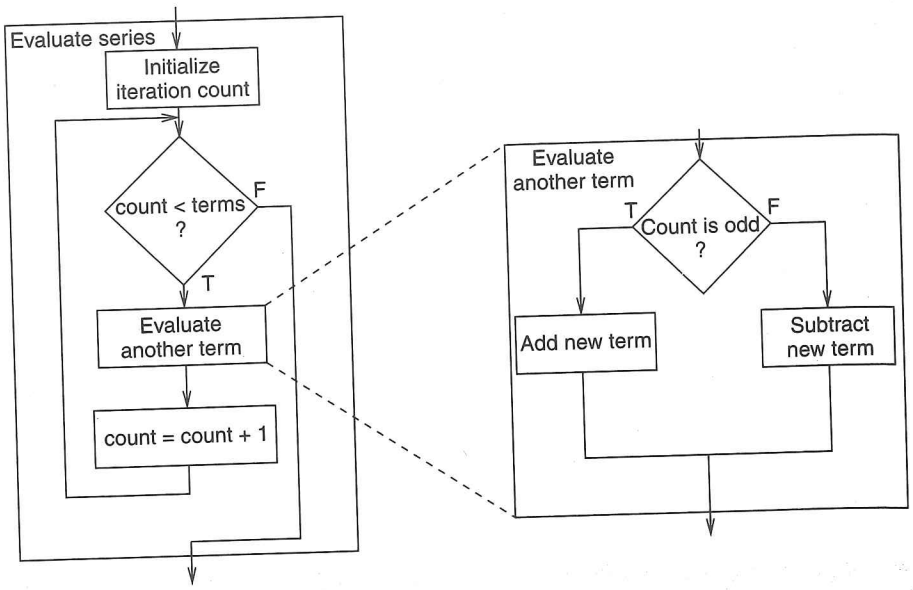


Figure 13.16 Incorporate the current term based on whether it is odd or even

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int count;          /* Iteration variable                */
6    int numOfTerms;     /* Number of terms to evaluate       */
7    double pi = 0;      /* approximation of pi               */
8
9    printf("Number of terms (must be 1 or larger) : ");
10   scanf("%d", &numOfTerms);
11
12   for (count = 1; count <= numOfTerms; count++) {
13     if (count % 2)
14       pi = pi + (4.0 / (2.0 * count - 1)); /* Odd term  */
15     else
16       pi = pi - (4.0 / (2.0 * count - 1)); /* Even term */
17   }
18
19   printf("The approximate value of pi is %f\n", pi);
20 }
```

Figure 13.17 A program to calculate π

13.4.2 Problem 2: Finding Prime Numbers Less than 100

Our next problem-solving example involves finding all the prime numbers that are less than 100. Recall that a number is prime only if the only numbers that evenly divide it are 1 and itself.

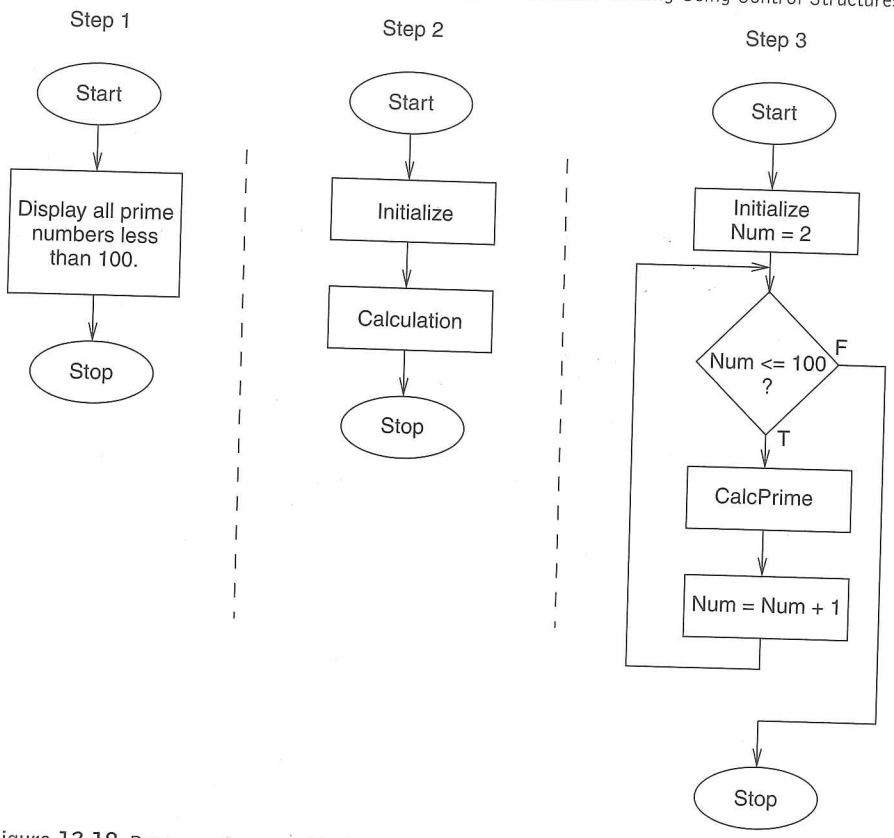


Figure 13.18 Decomposing a problem to compute prime numbers less than 100. The first three steps involve creating a loop that iterates between the 2 and 100

Step 0, as with our previous examples, is to select an appropriate data representation for the various data associated with the problem. Since the property of prime numbers only applies to integers, using the integer data type for the main computation seems a good choice.

Next we apply stepwise refinement to the problem to reduce it into a C program. We can approach this problem by first stating it as a single task (step 1). We then refine this single task into two separate sequential subtasks: Initialize and then perform the calculation (step 2).

Performing the *Calculation* subtask is the brunt of the programming effort. Essentially, the *Calculation* subtask can be stated as follows: We want to check every integer between 2 and 100 to determine if it is prime. If it is prime, we want to print it out. A counter-controlled loop should work just fine for this purpose. We can further refine the *Calculation* subtask into smaller subtasks, as shown in Figure 13.18. Notice that the flow diagram has the shape of a `for` loop.

Already, the problem is starting to resolve into C code. We still need to refine the *CalcPrime* subtask. In this subtask, we need to determine if the current number is prime or not. Here, we rely on the fact that any number between 2 and 100 that is *not* prime will have at least one divisor between 2 and 10 that is not itself. We

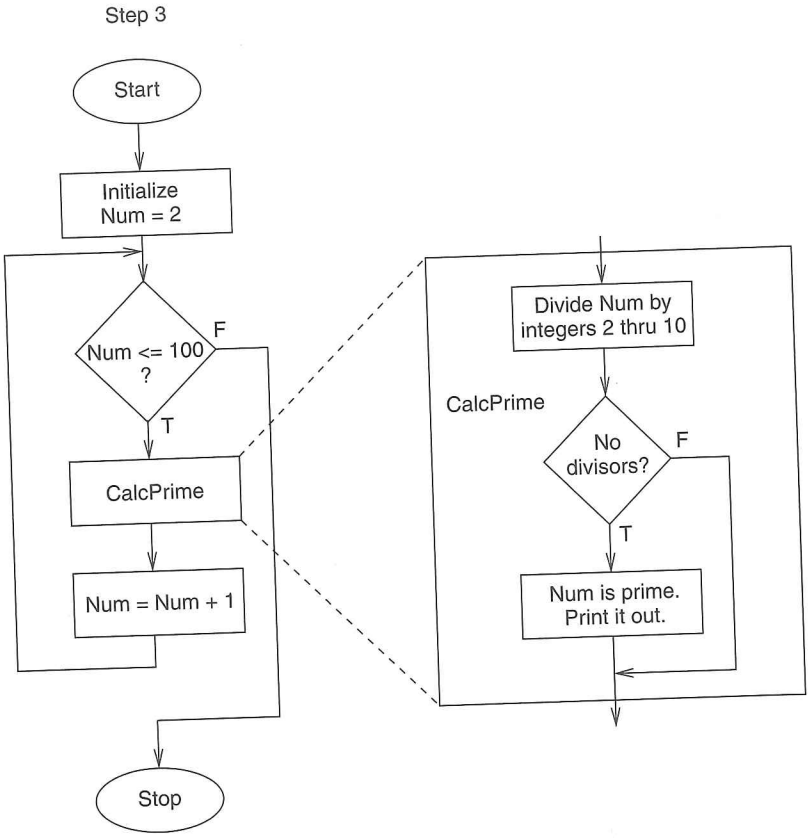


Figure 13.19 Decomposing the CalcPrime subtask

can refine this subtask as shown in Figure 13.19. Basically, we will determine if each number is divisible by an integer between 2 and 10 (being careful to exclude the number itself). If it has no divisors between 2 and 10, except perhaps itself, then the number is prime.

Finally, we need to refine the *Divide number by integers 2 through 10* subtask. It involves dividing the current number by all integers between 2 and 10 and determining if any of them evenly divide it. A simple way to do this is to use another counter-controlled loop to cycle through all the integers between 2 and 10. Figure 13.20 shows the decomposition using the iteration construct.

Now, coding this problem into a C program is a small step forward. The program is listed in Figure 13.21. There are two `for` loops within the program, one of which is nested within the other. The outer loop sequences through all the integers between 2 and 100; it corresponds to the loop created when we decomposed the *Calculation* subtask. An inner loop determines if the number generated by the outer loop has any divisors; it corresponds to the loop created when we decomposed the *Divide number by integers 2 through 10* subtask.

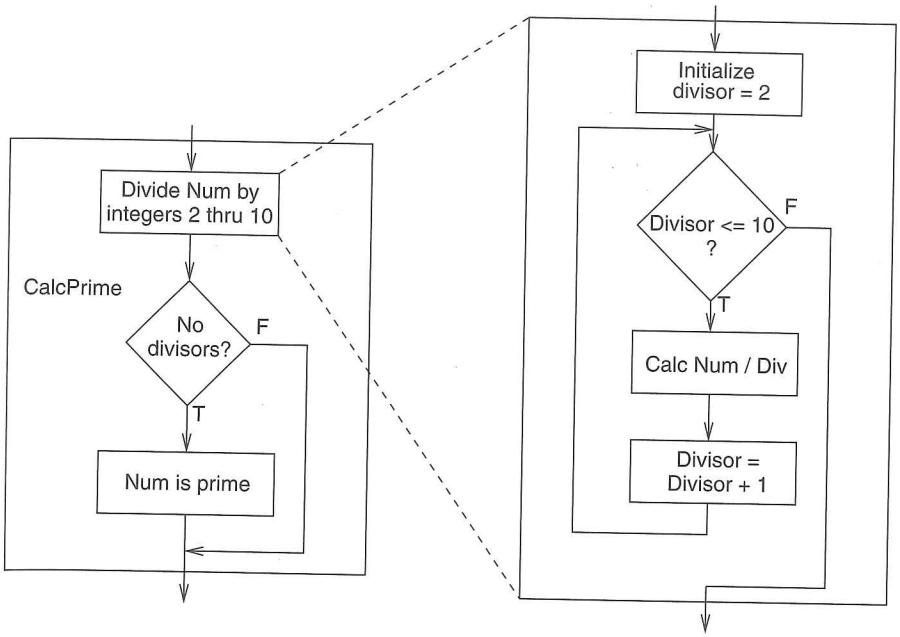


Figure 13.20 Decomposing the Divide numbers by integers 2 through 10 subtask

```
1  #include <stdio.h>
2  #define FALSE 0
3  #define TRUE  1
4
5  int main()
6  {
7    int num;
8    int divisor;
9    int prime;
10
11   /* Start at 2 and go until 100  */
12   for (num = 2; num <= 100; num++) {
13     prime = TRUE;    /* Assume the number is prime */
14
15     /* Test if the candidate number is a prime */
16     for (divisor = 2; divisor <= 10; divisor++)
17       if (((num % divisor) == 0) && num != divisor)
18         prime = FALSE;
19
20     if (prime)
21       printf("The number %d is prime\n", num);
22   }
23 }
```

Figure 13.21 A program that finds all prime numbers between 2 and 100

One item of note: If a divisor between 2 and 10 is found, then a flag variable called `prime` is set to *false*. It is set to *true* before the inner loop begins. If it remains *true*, then the number generated by the outer loop has no divisors and is therefore prime. To do this, we are utilizing the C preprocessor's macro substitution facility. We have defined, using `#define`, two symbolic names, `FALSE`, which maps to the value 0 and `TRUE`, which maps to 1. The preprocessor will simply replace each occurrence of the word `TRUE` in the source file with 1 and each occurrence of `FALSE` with 0.

13.4.3 Problem 3: Analyzing an E-mail Address

Our final problem in this section involves analyzing an e-mail address typed in at the keyboard to determine if it is of valid format. For this problem, we'll use a simple definition of validity: an e-mail address is a sequence of characters that must contain an at sign, "@", and a period, ".", with the at sign preceding the period.

As before, we start by choosing an appropriate data representation for the underlying data of the problem. Here, we are processing text data entered by the user. The type best suited for text is the ASCII character type, `char`. Actually, the best representation for input text is an array of characters, or *character string*, but as we have not yet introduced arrays into our lexicon of primitive elements (and we will in Chapter 16), we instead target our solution to use a single variable of the `char` type.

Next, we apply stepwise refinement. The entire process is diagrammed in Figure 13.22. We start with a rough flow of the program where we have two

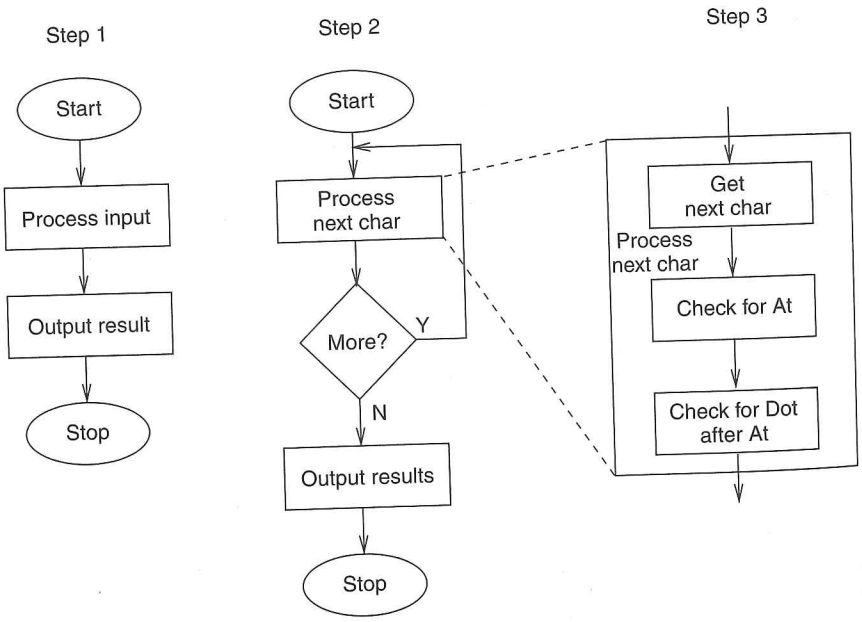


Figure 13.22 A stepwise refinement of the analyze e-mail address program

tasks (step 1): *Process input* and *Output results*. Here, the *Output results* task is straightforward. We will output either that the input text is a valid e-mail address or that it is invalid. The *Process input* task requires more refinement.

In decomposing the *Process input* task (step 2), we need to keep in mind that our choice of data representation (variable of the `char` type) implies that we will need to read and process the user's input one character at a time. We will keep processing, character by character, until we have reached the end of the e-mail address, implying that we select some form of sentinel-controlled loop. Step 2 of the decomposition divides the *Process input* task into a sentinel-controlled iteration construct that terminates when the end of an e-mail address is encountered, which we'll say is either a space or a newline character, `\n`.

The next step (step 3) of the decomposition involves detailing what processing occurs within the loop. Here, we need to check each character within the e-mail address and remember if we have seen an at sign or a period in the proper order. To do this, we will use two variables to record this status. When the loop terminates

```
1  #include <stdio.h>
2  #define FALSE 0
3  #define TRUE  1
4
5  int main()
6  {
7    char nextChar;    /* Next character in e-mail address */
8    int gotAt  = FALSE;  /* Indicates if At @ was found  */
9    int gotDot = FALSE;  /* Indicates if Dot . was found */
10
11   printf("Enter your e-mail address: ");
12
13   do {
14     scanf("%c", &nextChar);
15
16     if (nextChar == '@')
17       gotAt = TRUE;
18
19     if (nextChar == '.' && gotAt == TRUE)
20       gotDot = TRUE;
21   }
22   while (nextChar != ' ' && nextChar != '\n');
23
24   if (gotAt == TRUE && gotDot == TRUE)
25     printf("Your e-mail address appears to be valid.\n");
26   else
27     printf("Your e-mail address is not valid!\n");
28 }
```

Figure 13.23 A C program to determine if an e-mail address is valid

and we are ready to display the result, we can examine these variables to display the appropriate output message.

At this point, we are not far from C code. Notice that the loop structure is very similar to the flow diagram of the `do-while` statement. The C code for this problem is provided in Figure 13.23.

13.5 Additional C Control Structures

We complete our coverage of the C control structures by examining the `switch`, `break`, and `continue` statements. These three statements provide specialized program control that programmers occasionally find useful for very particular programming situations. We provide them here primarily for completeness; none of the examples in the remainder of the textbook use any of these three constructs.

13.5.1 The `switch` Statement

Occasionally, we run into programming situations where we want to perform a series of tests on a single value. For example, in the following code, we test the character variable `keyPress` to see if it equals a series of particular characters.

```
char keyPress;

if (keyPress == 'a')
   /* statement A */

else if (keyPress == 'b')
   /* statement B */

else if (keyPress == 'x')
   /* statement C */

else if (keyPress == 'y')
   /* statement D */
```

In this code, one (or none) of the statements labeled A, B, C, or D will execute, depending on the value of the variable `keyPress`. If `keyPress` is equal to the character *a*, then statement A is performed, if it is equal to the character *b*, then statement B is performed, and so forth. If `keyPress` does not equal *a* or *b* or *x* or *y*, then none of the statements are executed.

If there are many of these conditions to check, then many tests will be required in order to find the "matching" one. In order to give the compiler an opportunity to better optimize this code by bypassing some of this testing, C provides the `switch` statement. The following code segment behaves the same as the code in the previous example. It uses a `switch` statement instead of cascaded `if-else` statements.

```
char keyPress;

switch (keyPress) {
case 'a':
  /* statement A */
  break;

case 'b':
  /* statement B */
  break;

case 'x':
  /* statement C */
  break;

case 'y':
  /* statement D */
  break;
}
```

Notice that the `switch` statement contains several lines beginning with the keyword `case`, followed by a label. The program evaluates `keyPress` first. Then it determines which of the following `case` labels matches the value of `keyPress`. If any label matches, then the statements following it are executed.

Let's go through the `switch` construct piece by piece. The `switch` keyword precedes the expression on which to base the decision. This expression must be of integral type (for example, an `int` or a `char`). If one of the `case` labels matches the value of the expression, then program control passes to the statement or block associated with (usually, immediately below) that `case` label. Each `case` consists of a sequence of zero or more statements similar to a compound statement, but no delimiting braces are required. The place within this compound statement to start executing is determined by which `case` matches the value of the `switch` expression. Each `case` label within a `switch` statement must be unique; identical labels are not allowed.

Furthermore, each `case` label must be a constant expression. It cannot be based on a value that changes as the program is executing. The following is not a legal `case` label (assuming `i` is a variable):

```
case i:
```

In the preceding `switch` example, each `case` ends with a `break` statement. The `break` exits the `switch` construct and changes the flow of control directly to the statement after the closing brace of the `switch`. The `break` statements are optional. If they are not used, then control will go from the current `case` to the

next. For example, if the `break` after statement C were omitted, then a match on `case 'x'` would cause statement C *and* statement D to be executed. However, in practice, `case`s almost always end with a `break`.

We can also include a `default` case. This case is selected if the `switch` expression matches none of the `case` constants. If no `default` case is given, and the expression matches none of the constants, none of the `case`s are executed.

A stylistic note: The last `case` of a `switch` does not need to end with a `break` since execution of the `switch` ends there, anyway. However, including a `break` for the final `case` is good programming practice. If another `case` is ever added to the end of the `switch`, then you will not have to remember to also add the `break` to the previous `case`. It is good, defensive programming.

13.5.2 The `break` and `continue` Statements

In the previous section, we saw an example of how the C `break` statement is used with `switch`. The `break` statement, and also the `continue` statement, are occasionally used with iteration constructs.

The `break` statement causes the compiler to generate code that will prematurely exit a loop or a `switch` statement. When used within a loop body, `break` causes the loop to terminate by causing control to jump out of the innermost loop that contains it. The `continue` statement, on the other hand, causes the compiler to generate code that will end the current iteration and start the next. These statements can occur within a loop body and apply to the iteration construct immediately enclosing them. Essentially, the `break` and `continue` statements cause the compiler to generate an unconditional branch instruction that leaves the loop from somewhere in the loop body. Following are two example code segments that use `break` and `continue`.

```
/* This code segment produces the output: 0 1 2 3 4 */
for (i = 0; i < 10; i++) {
   if (i == 5)
      break;
   printf("%d ", i);
}
/* This code produces the output: 0 1 2 3 4 6 7 8 9 */
for (i = 0; i < 10; i++) {
   if (i == 5)
      continue;
   printf("%d ", i);
}
```

13.5.3 An Example: Simple Calculator

The program in Figure 13.24 performs a function similar to the calculator example from Chapter 10. The user is prompted for three items: an integer operand,

```
#include <stdio.h>

int main()
{
  int operand1, operand2;      /* Input values                */
  int result = 0;              /* Result of the operation    */
  char operation;              /* operation to perform       */

  /* Get the input values */
  printf("Enter first operand: ");
  scanf("%d", &operand1);
  printf("Enter operation to perform (+, -, *, /): ");
  scanf("\n%c", &operation);
  printf("Enter second operand: ");
  scanf("%d", &operand2);

  /* Perform the calculation */
  switch(operation) {
  case '+':
    result = operand1 + operand2;
    break;

  case '-':
    result = operand1 - operand2;
    break;

  case '*':
    result = operand1 * operand2;
    break;

  case '/':
    if (operand2 != 0)                    /* Error-checking code. */
      result = operand1 / operand2;
    else
      printf("Divide by 0 error!\n");
    break;

  default:
    printf("Invalid operation!\n");
    break;
  }

  printf("The answer is %d\n", result);
}
```

Figure 13.24 Calculator program in C

an operation to perform, and another integer operand. The program performs the operation on the two input values and displays the result. The program makes use of a `switch` to base its computation on the operator the user has selected.

13.6 Summary

We conclude this chapter by summarizing the key concepts we've covered. The basic objective of this chapter was to enlarge our set of problem-solving primitives by exploring the various control structures supported by the C programming language.

- **Decision Construct in C.** We covered two basic C decision statements: `if` and `if-else`. Both of these statements *conditionally* execute a statement depending on whether a specified expression is true or false.
- **Iteration Constructs in C.** C provides three iteration statements: `while`, `for`, and `do-while`. All of these statements execute a statement possibly multiple times until a specified expression becomes false. The `while` and `do-while` statements are particularly well-suited for expressing sentinel-controlled loops. The `for` statement works well for expressing counter-controlled loops.
- **Problem Solving Using Control Structures.** To our arsenal of primitives for problem solving (which already includes the three basic C types, variables, operators, and I/O using `printf` and `scanf`), we added control constructs. We practiced some problem-solving examples that required application of these control constructs.

Exercises

13.1 Recreate the LC-3 compiler's symbol table when it compiles the calculator program listed in Figure 13.24.

13.2 *a.* What does the following code look like after it is processed by the preprocessor?

```
#define VERO -2

if (VERO)
   printf("True!");
else
   printf("False!");
```

b. What is the output produced when this code is run?

c. If we modified the code to the following, does the code behave differently? If so, how?

```
#define VERO -2

if (VERO)
   printf("True!");
else if (!VERO)
   printf("False!");
```

13.3 An `if-else` statement can be used in place of the C conditional operator (see Section 12.6.3). Rewrite the following statement using an `if-else` rather than the conditional operator.

```
x = a ? b : c;
```

13.4 Describe the behavior of the following statements for the case when `x` equals 0 and when `x` equals 1.

a.
```
if (x = 0)
   printf("x equals 0\n");
else
   printf("x does not equal 0\n");
```

b.
```
if (x == 0)
   printf("x equals 0\n");
else
   printf("x does not equal 0\n");
```

c.
```
if (x == 0)
   printf("A\n");
else if (x != 1)
   printf("B\n");
else if (x < 1)
   printf("C\n");
else if (x)
   printf("D\n");
```

d.
```
int x;
int y;

switch (x) {
case 0:
   y = 3;

case 1:
   y = 4;
   break;

default:
   y = 5;
   break;
}
```

e. What happens if `x` is not equal to 0 or 1 for part 4?

13.5 Provide the LC-3 code generated by our LC-3 C compiler when it compiles the `switch` statement in part 4 of Exercise 13.4.

13.6 Figure 13.12 contains a C program with a nested `for` loop.

a. Mathematically state the series that this program calculates.

b. Write a program to calculate the following function:

$$f(n) = f(n-1) + f(n-2)$$

with the following initial conditions,

$$f(0) = 1, \quad f(1) = 1$$

13.7 Can the following `if-else` statement be converted into a `switch`? If yes, convert it. If no, why not?

```
if (x == 0)
   y = 3;
else if (x == 1)
   y = 4;
else if (x == 2)
   y = 5;
else if (x == y)
   y = 6;
else
   y = 7;
```

13.8 At least how many times will the statement called `loopBody` execute the following constructs?

a.
```
while (condition)
    loopBody;
```

b.
```
do
    loopBody;
while (condition);
```

c.
```
for (init; condition; reinit)
    loopBody;
```

d.
```
while (condition1)
    for (init; condition2; reinit)
        loopBody;
```

e.
```
do
    do
        loopBody;
    while (condition1);
while (condition2);
```

13.9 What is the output of each of the following code segments?

a.
```
a = 2;
while (a > 0) {
    a--;
}
printf("%d", a);
```

b.
```
a = 2;
do {
    a--;
} while (a > 0)
printf("%d", a);
```

c.
```
b = 0;
for (a = 3; a < 10; a += 2)
    b = b + 1;
printf("%d %d", a, b);
```

13.10 Convert the program in Figure 13.4 into one that uses a `switch` statement instead of `if-else`.

13.11 Modify the e-mail address validation program in Figure 13.23 so that it requires that at least one alphabetic character appears prior to the at sign, one appears between the at sign and the period, and one appears after the period in order for an e-mail address to be valid.

13.12 For the following questions, `x` is an integer with the value 4.

a. What output is generated by the following code segment?

```
if (7 > x > 2)
   printf("True.");
else
   printf("False.");
```

b. Does the following code cause an infinite loop?

```
while (x > 0)
   x++;
```

c. What is the value of `x` after the following code has executed?

```
for (x = 4; x < 4; x--) {
   if (x < 2)
      break;
   else if (x == 2)
      continue;
   x = -1;
}
```

13.13 Change this program so that it uses a `do-while` loop instead of a `for` loop.

```
int main()
{
   int i;
   int sum;

   for (i = 0; i <= 100; i++) {
      if (i % 4 == 0)
         sum = sum + 2;
      else if (i % 4 == 1)
         sum = sum - 6;
      else if (i % 4 == 2)
         sum = sum * 3;
      else if (i % 4 == 3)
         sum = sum / 2;
   }
   printf("%d\n", sum);
}
```

13.14 Write a C program that accepts as input a single integer `k`, then writes a pattern consisting of a single 1 on the first line, two 2s on the second line, three 3s on the third line, and so forth, until it writes `k` occurrences of `k` on the last line.

For example, if the input is 5, the output should be the following:

```
1
2  2
3  3  3
4  4  4  4
5  5  5  5  5
```

13.15 *a.* Convert the following `while` loop into a `for` loop.

```
while (condition)
   loopBody;
```

b. Convert the following `for` loop into a `while` loop.

```
for (init; condition; reinit)
   loopBody;
```

13.16 What is the output of the following code?

```
int r = 0;
int s = 0;
int w = 12;
int sum = 0;

for (r = 1; r <= w; r++)
   for (s = r; s <= w; s++)
      sum = sum + s;

printf("sum =%d\n", sum);
```

13.17 The following code performs something quite specific. Describe its output.

```
int i;

scanf("%d", &i);
for (j = 0; j < 16; j++) {
  if (i & (1 << j)) {
    count++;
  }
}

printf("%d\n", count);
```

13.18 Provide the output of each of the following code segments.

a.
```
int x = 20;
int y = 10;

while ((x > 10) && (y & 15)) {
   y = y + 1;
   x = x - 1;
   printf("*");
}
```

b.
```
int x;

for (x = 10; x ; x = x - 1)
   printf("*");
```

c.
```
int x;

for (x = 0; x < 10; x = x + 1) {
   if (x % 2)
      printf("*");
}
```

d.
```
int x = 0;
int i;

while (x < 10) {
  for (i = 0; i < x; i = x + 1)
    printf("*");
  x = x + 1;
}
```